SAND2019-9132C

Low-Power Deep Learning Inference using the SpiNNaker Neuromorphic Platform

Craig M. Vineyard, Ryan Dellana, James B. Aimone, Fredrick Rothganger, William M. Severa
Sandia National Laboratories
Albuquerque, NM
cmviney@sandia.gov,rdellan@sandia.gov,jbaimon@sandia.gov,frothga@sandia.gov,wmsever@sandia.gov

ABSTRACT

With the successes deep neural networks have achieved across a range of applications, researchers have been exploring computational architectures to more efficiently execute their operation. In addition, to the prevalent role of graphics processing units (GPUs), many accelerator architectures have emerged. Neuromorphic is one such particular approach which takes inspiration from the brain to guide the computational principles of the architecture including varying level of biological realism. In this paper we present results on using the SpiNNaker neuromorphic platform (48-chip model) for deep learning neural network inference. We use the Sandia National Laboratories developed Whetstone spiking deep learning library to train deep multi-layer perceptrons and convolutional neural networks suitable for the spiking substrate on the neural hardware architecture. By using the massively parallel nature of SpiNNaker, we are able to achieve, under certain network topologies, substantial network tiling and consequentially impressive inference throughput. Such high-throughput systems may have eventual application in remote sensing applications where large images need to be chipped, scanned, and processed quickly. Additionally, we explore complex topologies that push the limits of the SpiNNaker routing hardware and investigate how that impacts mapping software-implemented networks to on-hardware instantiations.

1 INTRODUCTION & BACKGROUND

Historical advances in computational capabilities have been enabled by trends such as Moore's Law and Dennard Scaling. With these scaling laws coming to an end, rather than exponential advances in manufacturing capability furthering general purpose architectures, instead specialized architectures are being considered. In particular, the growing popularity of deep neural networks and machine learning is fueling research into accelerators better suited for these workloads [2].

With the proliferation of machine learning accelerators and neuromorphic architectures, there has been a lag in the development of the software stack to interface to these emerging devices. CUDA serves as an application programming interface to Nvidia GPUs [11], however many other architectures lack this capability necessitating lower level programming efforts to make use of the specialized hardware. Efforts are emerging to try and address this need. For example, the Neural Network Exchange Format provides a specification in which emerging architectures can interface to by providing parsing and compilation tools specific to their architecture [5]. Similarly, Glow developed by Facebook strives to provide an intermediate representation of high level machine learning representations for which hardware targets can support [10].

However, these and other emerging efforts are not presently targeting spiking neuromoprhic hardware, and additionally the deep neural network (DNN) algorithms themselves need to either learn spiking representations or be converted to map on to neuromorphic hardware.

Developed by Sandia National Laboratories, Whetstone is a method for training deep artificial neural networks for binary communication [14]. The Whetstone method, offers an iterative approach that integrates well with a variety of neural network topologies and integrates with standard deep learning libraries (namely Keras on Tensorflow). The Whetstone method is illustrated in Fig. 1. This approach enables rapid prototyping and training of deep spiking neural networks that are neuromorphic-ready at a variety of scaled and complexities. Using Whetstone to target neuromorphic architectures, in this article we present results exploring the efficacy of using the University of Manchester SpiNNaker architecture for low-power deep learning inference.

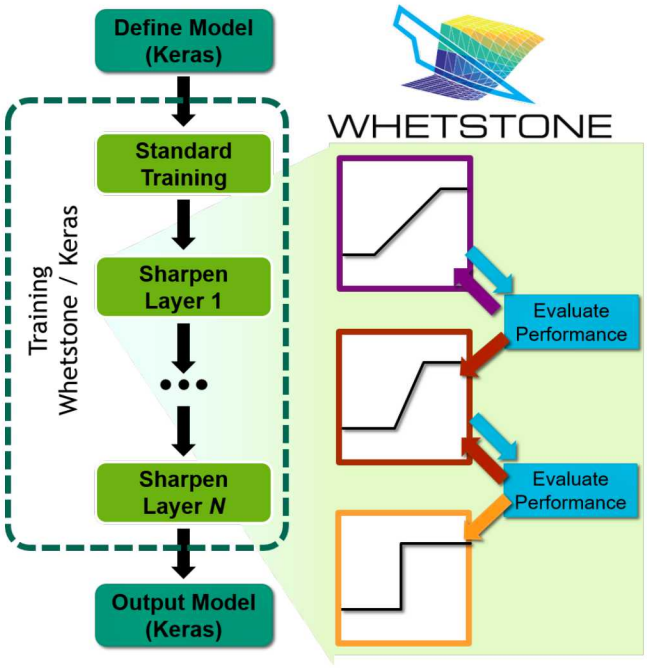

Figure 1: Illustration of the Whetstone Method

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

2 APPROACH

The SpiNNaker architecture is a scalable, energy-efficient, massively parallel computing architecture inspired by the function and structure of the brain [3, 4]. The fundamental unit of the SpiNNaker architecture is a multicore System-on-Chip comprised of lightweight ARM968 processor cores each with local instruction and data memory, and a packet router. Multi-cast routing via address event representation both allows cores within each chip to communicate as well as chips within the overarching architecture. A communication mesh allows spikes from individual neurons to route to any other neuron in the system. For our work here, we have used a 48-node (864 core) SpiNNaker board.

For mapping applications onto the SpiNNaker infrastructure, low level C or Python code may be used as well as high-level neural description languages such as PyNN [1] and Nengo [15]. SpyNNaker provides a software package for simulating PyNN defined networks on the SpiNNaker platform [9]. Alternatively, here we explore using the Whetstone method to generate binary-activation quantized ANNs as an additional approach to mapping onto the SpiNNaker neuromorphic platform complementing the other existing approaches.

2.1 Mapping

Given the available software interfaces to the SpiNNaker platform, we chose to implement Whetstone networks using SpyNNaker. This interface provides a higher-level abstraction compared to the C interface, and we expect, going forward, that SpyNNaker will provide greater interoperability with a range of peripheral libraries (data loaders, spiking sensors, etc.). Given the flexibility of connections, most components and parameters of the Whetstone network map directly to the SpyNNaker network. However, as follows are three settings which must be addressesd to map Whetstone networks onto SpiNNaker:

(1) **Bias** Most deep neural network use a bias that helps improve separability away from the origin. In a Whetstone network, the bias is trained as with any other weight parameter. However, in practice, the bias is 'rolled' into the threshold of the neuron. For SpyNNaker networks, this implies a different threshold for each neuron, which by default, would require each neuron to be on separate cores. We alleviate this problem by either (a) training a network without bias or (b) using a separate neuron to relay a bias signal (via weights). This signal allows the neurons to each have an effective threshold without changing the threshold parameter.

(2) **Neuron Types** Whetstone networks require very simple neurons, essentially perceptrons. In contrast, SpyNNaker (and SpiNNaker) can support much more complicated neuron models. We use the IF_curr_delta neuron model as our baseline, however, we have also created a custom neuron type IF0_curr_delta which instantaneously decays. While relaxing biological realism, the IF0_curr_delta neuron is advantageous in enabling faster firing rates and can also enable denser neuron packing on the SpiNNaker board.

(3) **Weight Precision** Since Whetstone uses standard deep learning methods, the weights are floating point parameters (either 16-bit or 32-bit). However, most neuromorphic platforms use some fixed-point representation; SpiNNaker uses $Q15.16$ integer values. As such, we decided to investigate the impact of reduced fixed-point precision weights. In general, we have seen that Whetstone networks remain effective, with essentially no loss in performance, down to $Q4.8$.

2.2 From Tensors to SpiNNaker

The Whetstone-based mapping from tensor representations to neuron represenations involves six transformations, with a seventh (time-division multiplexing) optional step. Figure 2 illustrates these transformation steps, each of which is described subsequently.

(1) **Binary Quantizing Activations** Spiking neural networks are a form of event-based computing. A point of interesection between the event-based paradigm and traditional artificial neural networks is the binary-activation quantized ANN, which allows activations to be interpreted as firing events. Whetstone [13] produces such binary-ANNs by gradual modification (i.e. sharpening) of activation functions during training until they become binary threshold gates. In order to allow convergence this process is performed layer-wise bottom-up using an adaptive algorithm which seeks to minimize increases in training loss.
One stipulation of Whetstone is that activation functions be "sharpenable", meaning they support a parameter that can be used to set the steepness of their nonlinearity in a continuous space between the original function and the step/threshold function. Currently, this means using either a parameterized sigmoid or a bounded-RELU (bRELU). bRELUs have been shown to be as effective or nearly as effective as RELUs, and the bounded range allows them to be easily converted to a spiking threshold function [7]. We parameterize our units as

$$h_{\alpha,\beta} = \begin{cases} 1, & \text{if } x_i \geq \beta \\ (x_i - \alpha)/(\beta - \alpha) & \text{if } \alpha \leq x_i < \beta \\ 0, & \text{if } x_i \leq \alpha, \end{cases} \tag{1}$$

and assert that $|\beta - 0.5| = |\alpha - 0.5|$. With $\alpha = 0$ and $\beta = 1$, $h_{\alpha,\beta}$ is a standard bounded RELU. However, as $\alpha \to 0.5$, $h_{\alpha,\beta} \to h$. After an initial period of conventional training, the spiking bRELUs are sharpened by reducing the difference between α and β. The rate and method of convergence can be determined either prior to training or dynamically during training.

(2) **Binary Coding Inputs and Outputs** Recoding floating-point inputs and outputs to binary-representations can present challenges. Consider a 1-hot output encoding scheme often used in classification. A 1-hot encoding is used for labeling classes where an object belonging to class k of n total classes is represented by an n-length vector with 1 at the k'th element and the remainder being 0. This 0-1 encoding is actually a form of unary coding but is compatible with binary activation neurons. In practice some output neurons do not survive the Whetstone sharpening process (a.k.a. the "dead node" problem). This issue is addressed by using redundant output neurons for each class. During training, the number of active neurons for a given class is summed to produce the logits which pass into a softmax layer to allow

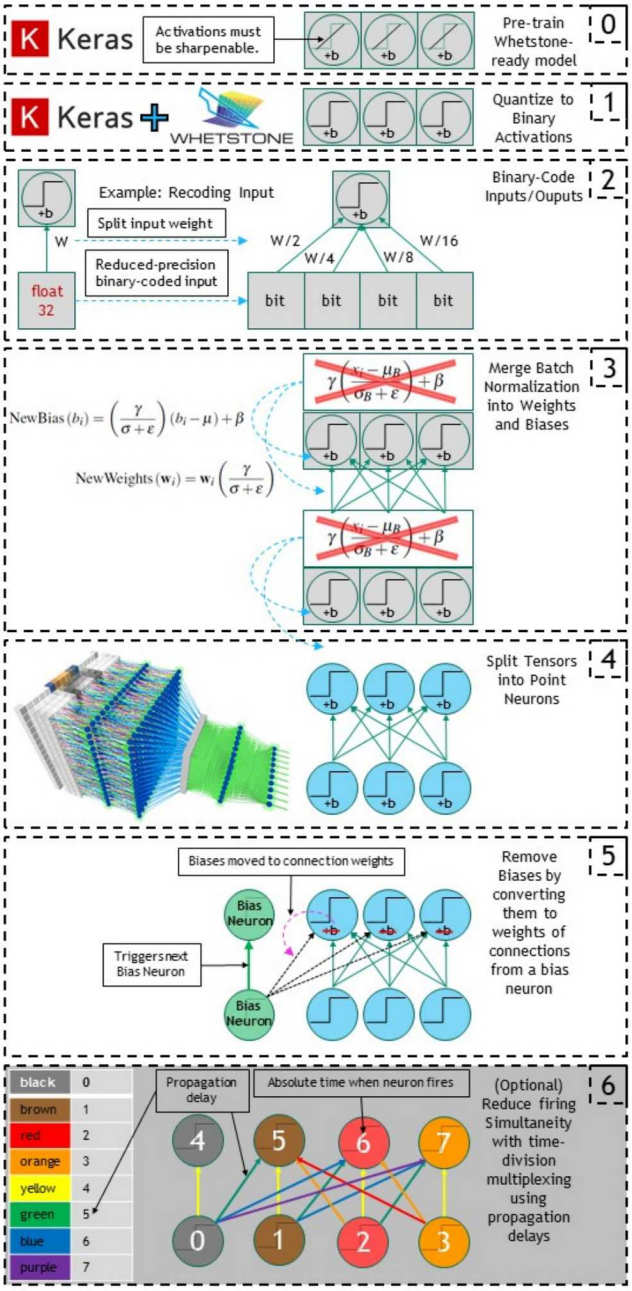

Figure 2: Illustration of each of the steps applied in our approach to using the Whetstone method to map neural networks onto to the SpiNNaker neuromorphic architecture. Steps 0-5 are essential with step 6 optional. Not shown is a seventh step of mapping the resulting Whetstone compatible SpiNNaker network onto the hardware through the PyNN interface.

for the use of cross-entropy loss functions. After training, the softmax layer is discarded and replaced with a majority voting scheme which operates directly on the binary n-hot outputs.

To approximate continuous outputs, a more direct unary coding scheme can be used, where the percentage of active neurons can be mapped to some floating point range such as [0.0, 1.0], though this is innefficient. Input encoding is actually much easier, since it allows floating point inputs to be recoded directly to binary by splitting out the single input channel into bit channels, and logarithmically partitioning the original weights over the binary input dimensions in decreasing order of bit significance.

(3) **Removing Batch Normalization** The use of batch normalization has been shown to significantly improve stability during Whetstone sharpening [13]. However, in Tensorflow batch normalization layers persist after training, taking the form of linear transforms (Equation 2) consisting of four parameters for each neuron (and each convolution kernel).

$$BN\left(x_i\right) = \gamma\left(\frac{x_i - \mu_B}{\sigma_B + \epsilon}\right) + \beta \tag{2}$$

Gamma and beta are directly optimized during training while mu and sigma are computed using the moving averages of pre-activation means and variances over the batches.

$$\text{NewWeights}\left(\mathbf{w}_i\right) = \mathbf{w}_i\left(\frac{\gamma}{\sigma + \epsilon}\right) \tag{3}$$

$$\text{NewBias}\left(b_i\right) = \left(\frac{\gamma}{\sigma + \epsilon}\right)\left(b_i - \mu\right) + \beta \tag{4}$$

Before we break out the tensors into point neurons, the model is simplified by merging these parameters into the weights and biases using 3 and 4. Biases are automatically added for layers that do not use them natively. Note this method requires batch normalization layers be applied before the nonlinearity rather than after it.

(4) **Breaking Tensors into Connection Tuples** As an intermediate sparse representation between tensors and neurons, we use a simple list of weighted directed edges represented as 3-tuples (presynapic-neuron-id, postsynaptic-neuron-id, connection-weight). For dense layers, each row of the weight matrix maps directly to the connection weights of a single unique neuron. In the case of convolutions this involves creating a unique vertex for each position of each convolution kernel, and omitting connections where there was previously zero padding. Max-pooling layers are similar to convolution, but with no padding and a step size equal to the receptive field width. In a binary-activation quantized net, max-pooling essentially becomes a multi-input OR-gate, so the only constraint on connection weights is that they each individually be sufficient to induce the max-pooling neuron to fire. Average-pooling currently is not supported.

(5) **Replacing Biases with Bias Neurons** Biases can be interpreted as either firing thresholds or as the weights on an additional input connection from a bias neuron. In PyNN, an entire population must share the same neuron template which includes the firing threshold, so we use the bias neuron interpretation. Bias neurons are added to each population

and daisy-chained with a strong excitatory connection so that each triggers the bias neuron of the next layer.
To accomplish this using only one bias neuron per layer, it is easier to perform this step after converting to the sparse representation. While it is technically possible to introduce a single bias neuron per layer within tensor-based frameworks, convolution layers make this inelegant.

(6) **Time-division Multiplexing (Optional)** Binary activation quantized networks running in spiking hardware allow "single-pass inference", where multiple samples can pass through the net simultaniously as separate "wave-fronts". This provides advantages in throughput and latency relative to rate-coded methods which must accumulate spikes for some period of time [6]. However, binary-activation ANNs require global synchrony to produce correct one-shot output, and their real-time simulation produces synchronous bursts of activity which can overload the communication fabric, possibly breaking global synchrony.
To reduce this congestion, we use the propagation delay ring-buffers for time-division multiplexing of the routers. While SpiNNaker processes synaptic events asynchronously, neuron state updates are local clock-driven synchronous, which we take advantage of to approximate global synchrony. When a firing event is generated by a neuron, SpiNNaker immediately transmits the spike packet to the destination cores where it waits in a ring buffer for a time determined by the propagation delay. Rather than having all neurons of a presynaptic population fire concurrently, we stagger their firing and use the delays to ensure all synaptic events from the source population induce a membrane potential at the correct time-increment. Thus, while firings of source population neurons are not synchronous, their effects downstream are.

To map the resulting Whetstone network onto SpiNNaker we use the PyNN interface by converting the DNN layers to PyNN populations. At this point the weights are still stored in floating-point precision, with the conversion to the target precision is handled in the translation to SpyNNaker. For SpiNNaker 1.0, this involves reducing the precision to Q15.16.

3 RESULTS

We have explored implementing small dense multi-layer neural networks as well as a convolutional neural network distributed across the SpiNNaker 48 platform. Given the scale of the platform and the small footprint of basic MLPs, we anticipate an optimal 'accuracy/throughput' configuration to involve network tiling—where multiple copies of the network are instantiated across different chips/cores and multiple inputs are fed simultaneously onto the board. This configuration is compatible with many image-scanning applications such as those involving satellite imagery and remote area mapping.

During testing, we found that having a large number of neurons ($\approx$ 400) spike simultaneously within a core overwhelms the SpiNNaker router, causing errors. While a small number of dropped or delayed spikes may be tolerable for the network, we still wanted to develop methods to mitigate this hardware-specific issue. For any layer configuration (i.e., convolution or densely connected), we can use the flexible delays supported in-hardware to temporally demux the signal, see Fig. 3. More specifically, we split a layer into different temporal groups each of which have outbound (axon-side) synapses of a different delay. Then, post-synaptic temporal groups integrate signals distributed in time, which lowers the number of spikes generated at any given timestep. However, care must be taken to ensure that the correct spikes are delivered to downstream neurons in-sync. For convolutional layers, we recognize that the connectivity is spatially oriented. As such, we can distribute neurons across cores so that nearby neurons (regardless of layer) are located in the same core. In the experiments presented here, this approach improved communication bottlenecks, but required careful layout.

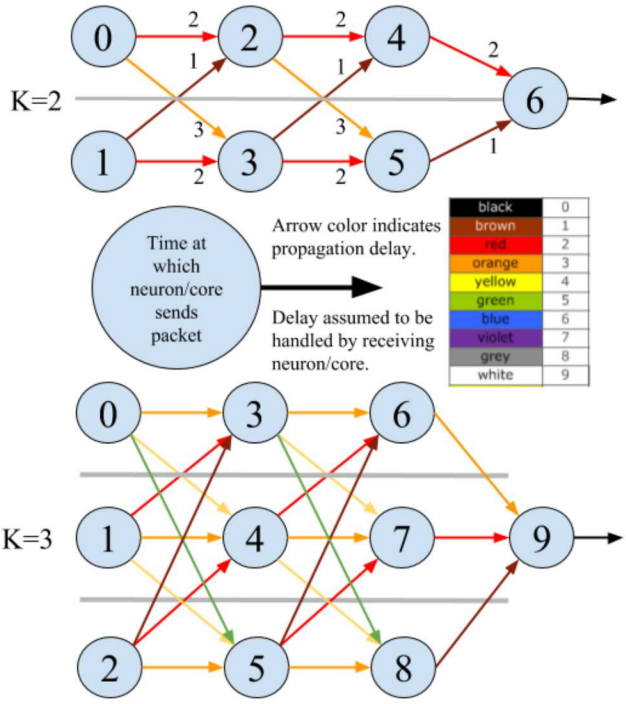

Figure 3: Depiction of the temporal distribution of spikes to avoid communication overloads on SpiNNaker

Table 1 provides the results and paramaterization of applying three different neural network topologies to the Binary Mnist dataset. The network toplogies are as follows with 10-hot output encoding:

(1) Small MLP: 785 → 100 → 100 → 100
(2) Medium MLP: 785 → 800 → 800 → 800 → 100
(3) Convolution Network: 785 → conv1 (5x5 x32 kernels) → maxpool (2,2) → conv2 (5x5 x64 kernels) → maxpool (2,2) → 500 dense → 100

These network topologies were selected through an iterative exploration and rapid prototyping of networks which converged

with decent accuracy and were compatible with SpiNNaker based upon architectural observations which follow.

3.1 Discussion

SpiNNaker is designed to run networks within biological timing and topological constraints, which are often at odds with the demands placed on the hardware by synchronous single-pass networks. In the following, we take a closer look at the constraints on synchrony, propagation delays, and intercore connectivity, all of which have significant performance impact.

- **Intercore Connectivity Constraints** Both neuron counts and fan-in are typically high in DNNs. The point-neuron interpretation of convolution requires a number of neurons equal to the size of the output tensor, and the projections between relatively small convolutional layers easily result in fan-ins exceeding 255 (ex. fan-in of 288 between two 32-filter conv-2d layers with 3x3 receptive fields).
  While it is technically possible to place more than 255 neurons on a given SpiNNaker core, in cases where there is maximum fan-in greater than 255, the current firmware imposes a 255 neuron/core limit. Specifically, a given neuron may project to a maximum of 255 neurons in another core; due to the 8-bit row-length limit in the synnaptic matrix (as imposed by the Address List) [9]. It is important to note that SpyNNaker divides application vertices (i.e. PyNN populations) into machine vertices which will actually fit in the cores, so these limits do not actually apply to pyNN population sizes, but rather the partitioned sub-populations (i.e. machine vertices). This, in turn, is set by calling “set_number_of_neurons_per_core" through the SpyNNaker python API.
- **Delay Overhead and Limits** According to [9], propagation delays greater than 16 time-steps are too long for the ring buffers to maintain and require the use of the “DelayExtensionVertex" application. In this scheme, each neuron is given a delay extension neuron that intercepts spikes and holds them for a longer delay period before relaying them to the specified destinations, at the cost of routhly doubling the required neurons and cores.
  Contrary to the above, our experiments found 10 time-steps (rather than 16) to be the maximum supported delay without invoking the DelayExtensionVertex. In our time-division multiplexing scheme, K temporal groups produce a maximum delay of (K*2 - 1), allowing us to use up to 5 temporal groups without invoking the DelayExtensionVertex. However, after this initial invocation, no additional cores are required up the the maximum supported delay of 144 time-steps [9], theoretically allowing the use of up to 72 temporal groups.
  Besides a doubling of core usage, we also observed the use of delay extension vertices to be associated with a significant increase in the incidence of dropped event packets. For example, in Table 1 it can be seen that the convolutional network (using 20 temporal groups) required a time-scale-factor of 14 to completely avoid dropped packets. We suspect this extra processing time is needed in order to search the larger master population tables which are required to accomodate the delay extention neurons (see section 3.4.2 of [9]).
- **No Guarantee of Global Synchrony** SpiNNaker is locally synchronous but globally asynchronous, meaning that synaptic events are handled asynchronously as soon as they arrive, while the neuron compartment models are updated on a clock. However, because SpiNNaker strives to achieve real-time performance, each core must drop unprocessed packets when it is unable to finish processing synaptic events prior to a clock tick. Because Whetstone networks are essentially threshold gates which evaluate inputs in a single time-step, dropped packets can significantly degrade network performance. While results vary, some of our experiments show that dropped packets can result from as few as 400 simultaneous events sent from one core to another. And, while SpiNNaker does support a packet rejection mechanism, it can only handle one packet before dropping packets. Thus, our deployment of Whetstone nets on SpiNNaker is limited by the SpiNNaker clock rate (200MHz).
  Using the current firmware, processing a single synaptic events takes 23 ARM968 instructions (20 is theoretical optimal) [9]. SpiNNaker can ideally handle about 5,000 synaptic events per core per millisecond. Therefore, at the current max of 255 neurons/core, the average number of synaptic events per neuron that can be handled in 1 millisecond is about 20. So, without any guarantees of activity sparsity, we can only afford a fan-in of 20. To achieve higher fan-in requires slowing down the simulation proportionally, or constraining worst-case activity sparsity proportionally. Typical convolution fan-in ranges from about 288 to 1600, requiring a slowdown somewhere between 15x to 80x, yielding throughput of between 66 and 12.5 fps. However, in reality, SDRAM contention-induced latency slows this down further. While it would seem these issues could be handily solved by increasing the time-scale-factor, as explained in section 2.2, we have found the synchronous nature of the events produced by Whetstone networks to overwhelm the communications fabric. It is for this reason we had to devise time-division multiplexing.

4 CONCLUSIONS

The SpiNNaker architecture has been developed with the goal of modeling large spiking neural networks with connectivity similar to the brain and capable of operating in biological real time [3]. Accordingly, even though the SpiNNaker architecture was not designed for ANN and DNN execution explicitly, our results show the impact of Whetstone trained binary communication neural networks exploiting the parallelism of the SpiNNaker neuromorphic platform. Previously, Serrano-Gotarrendona et al. used temporal coding to put similar sized convolutional networks on SpiNNaker [12]. And the recent work by Liu et al. explores putting deep neural networks on the SpiNNaker 2 prototype introducing DEEP R as a method to train an algorithm on board by continuously re-writing the network [8]. Our approach utilizing the Whetstone method provides an alternative method able to attain high throughput by utilizing binary communication rather than temporal coding. In

Table 1: Performance of parallel networks on SpinNNaker 48

Network	Small MLP	Medium MLP	Convolution Network
Total Neurons	57000	72500	47640
Total Cores	760	754	371
Total Chips Utilized	48	48	28
Network Tiles	190	29	1
Timescale Factor	5.0	6.0	14.0
Temporal Groups	0	0	20
Sample Delay (ms)	2	2	28
Throughput (frames/sec)	15317	2340	3.25
Accuracy	94%	97.7%	98.1%

doing so we are also able to achieve high classification accuracy relative to comparable spiking neuromorphic approaches while minimizing temporal delays.

By distributing many copies of a network across the SpiNNaker 48 platform, this highlights the potential power to implement an ensemble of networks in parallel or to alternatively partition the processing of a larger aggregate image into smaller more manageable tiles. In this latter case, rather than having a monolithic DNN operating upon a large image, such as those generated by remote sensing satellites, smaller tiles from the overarching scene can be processed in parallel. For example, rather than 190 or 29 network tiles each processing different binary MNIST inputs as we have shown here, instead they could be sub-samples of a larger image. This shows exciting promise for a large neuromorphic platform to help enable real time processing of sensor data.

Not only have our scaling studies highlighted the potential of general purpose neuromorphic architectures, but additionally have helped identify implications of workload impacts upon architectural choices. As SpiNNaker is designed for biological real time operation, we observed simultaneous concurrent local spike routing can overwhelm the routing fabric. While biological systems typically operate upon sparser activity, the dense spiking of MLP and convolution networks identify there are tradeoffs to the overhead of multi-cast asynchronous routing compared with a synchronous approach. In the case of the former, with sparse activity it can be advantageous to only route spikes via packets carrying address event representation data. However, once this activity saturates the routing fabric with dense activity an architecture may be more optimally implemented with synchronous communication. The optimal design choice depends upon the characteristics of the application workloads.

Overall, these results highlight the potential of highly parallel spiking neuromorphic architectures to enable computations such as inference efficiently, even though that is not a use case the architecture was designed for, showing the great potential of neural-inspired approaches to computing.

ACKNOWLEDGMENT

This work was supported by the Advanced Simulation and Computing, and the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

REFERENCES

[1] Brüderle, D., Müller, E., Davison, A. P., Muller, E., Schemmel, J., and Meier, K. Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system. *Frontiers in neuroinformatics 3* (2009), 17.

[2] Dean, J., Patterson, D., and Young, C. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro 38*, 2 (2018), 21–29.

[3] Furber, S. Large-scale neuromorphic computing systems. *Journal of neural engineering 13*, 5 (2016), 051001.

[4] Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., and Brown, A. D. Overview of the spinnaker system architecture. *IEEE Transactions on Computers 62*, 12 (2013), 2454–2467.

[5] Group, T. K. N. W. Neural network exchange format version 1.0 revision 3, 2018.

[6] Hunsberger, E., and Eliasmith, C. Training spiking deep networks for neuromorphic hardware. *arXiv preprint arXiv:1611.05141* (2016).

[7] Liew, S. S., Khalil-Hani, M., and Bakhteri, R. Bounded activation functions for enhanced training stability of deep neural networks on visual pattern recognition problems. *Neurocomputing 216* (2016), 718–734.

[8] Liu, C., Bellec, G., Vogginger, B., Kappel, D., Partzsch, J., Höppner, S., Maass, W., Furber, S. B., Legenstein, R., Mayr, C. G., et al. Memory-efficient deep learning on a spinnaker 2 prototype. *Frontiers in Neuroscience 12* (2018), 840.

[9] Rhodes, O., Bogdan, P. A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., Lester, D. R., Mikaitis, M., Plana, L. A., Rowley, A. G., et al. spynnaker: a software package for running pynn simulations on spinnaker. *Frontiers in Neuroscience 12* (2018), 816.

[10] Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Nadathur, S., Olesen, J., et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[11] Sanders, J., and Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[12] Serrano-Gotarredona, T., Linares-Barranco, B., Galluppi, F., Plana, L., and Furber, S. Convnets experiments on spinnaker. In *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on* (2015), IEEE, pp. 2405–2408.

[13] Severa, W., Vineyard, C. M., Dellana, R., Verzi, S. J., and Aimone, J. B. Whetstone: A method for training deep artificial neural networks for binary communication. *arXiv preprint arXiv:1810.11521* (2018).

[14] Severa, W., Vineyard, C. M., Dellana, R., Verzi, S. J., and Aimone, J. B. Training deep neural networks for binary communication with the whetstone method. *Nature Machine Intelligence 1*, 2 (2019), 86.

[15] Stewart, T. C., and Eliasmith, C. Large-scale synthesis of functional spiking neural circuits. *Proceedings of the IEEE 102*, 5 (2014), 881–898.